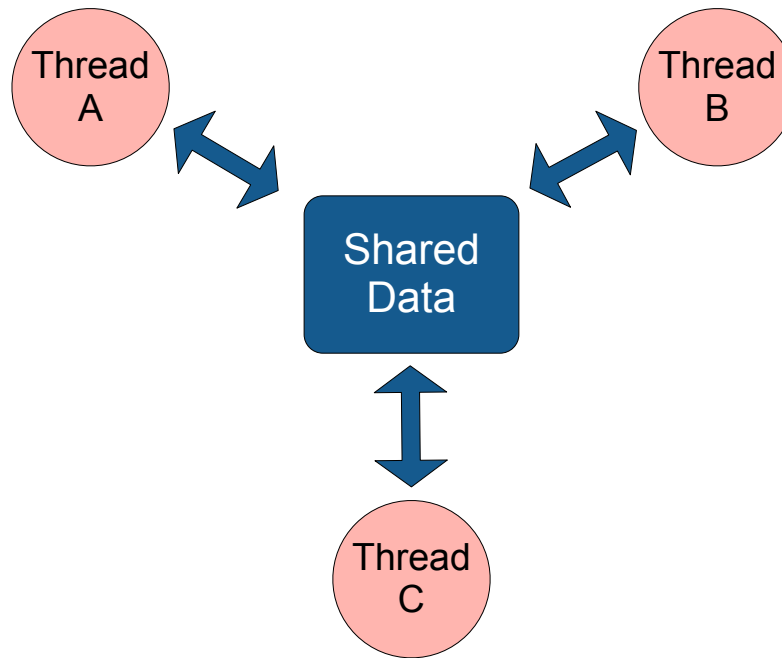


Synchronization



ATM Program

```
// get funds in account
int total = get_account_total();

// get amount to withdraw
int withdraw = get_withdraw_amount();

// check for sufficient funds
if (total >= withdraw) {

    // withdraw funds from account
    update_account_total(total - withdraw);
    dispense_money(withdraw);

}
```

Too Much Milk

```
You      if (noMilk) {
           buy milk;
           }

           if (noMilk) {
           buy milk;
           } Your Roommate
```

Time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery store	
3:15		Arrive home
3:20	Arrive at grocery store	Look in fridge, no milk
3:25	Buy milk	Leave for grocery store
3:30	Arrive home, put milk in fridge	
3:35		Arrive at grocery store
3:40		Buy milk
3:45		Arrive home – too much milk!

Too Much Milk: Solution 1?

Thread A

```
if (noMilk & noNote) {
    leave note;
    buy milk;
    remove note;
}
```

Thread B

```
if (noMilk & noNote) {
    leave note;
    buy milk;
    remove note;
}
```

Too Much Milk: Solution 2?

Thread A

```
leave note A;  
if (noNote B) {  
    if (noMilk) {  
        buy milk;  
    }  
}  
remove note A;
```

Thread B

```
leave note B;  
if (noNote A) {  
    if (noMilk) {  
        buy milk;  
    }  
}  
remove note B;
```

Too Much Milk: Solution 3?

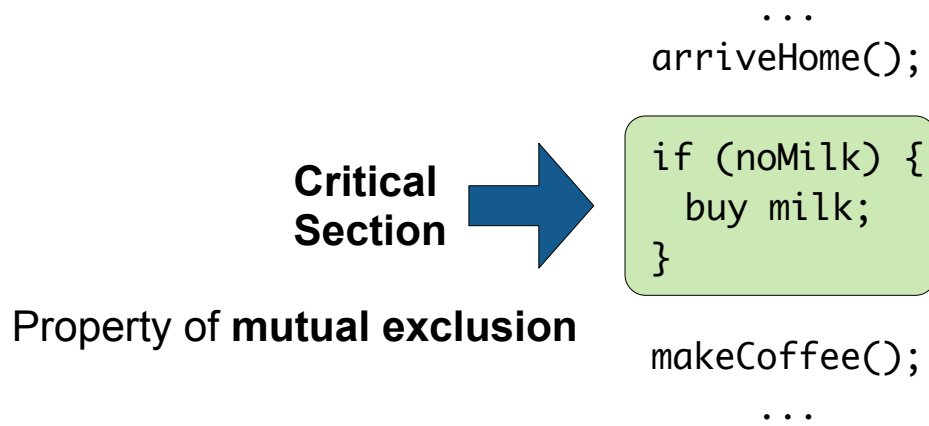
Thread A

```
leave note A;  
while (note B) {  
    do nothing;  
}  
if (noMilk) {  
    buy milk;  
}  
remove note A;
```

Thread B

```
leave note B;  
if (noNote A) {  
    if (noMilk) {  
        buy milk;  
    }  
}  
remove note B;
```

Critical Sections



Mutex Locks



Too Much Milk with Locks

Thread A

```
lock.acquire();  
if (noMilk) {  
    buy milk;  
}  
lock.release();
```

Thread B

```
lock.acquire();  
if (noMilk) {  
    buy milk;  
}  
lock.release();
```

Implementing Locks: Interrupts (version 1)

```
class Lock {  
    public:  
        void acquire();  
        void release();  
}
```

```
Lock::acquire() {  
    disable interrupts;  
}
```

```
Lock::release() {  
    enable interrupts;  
}
```

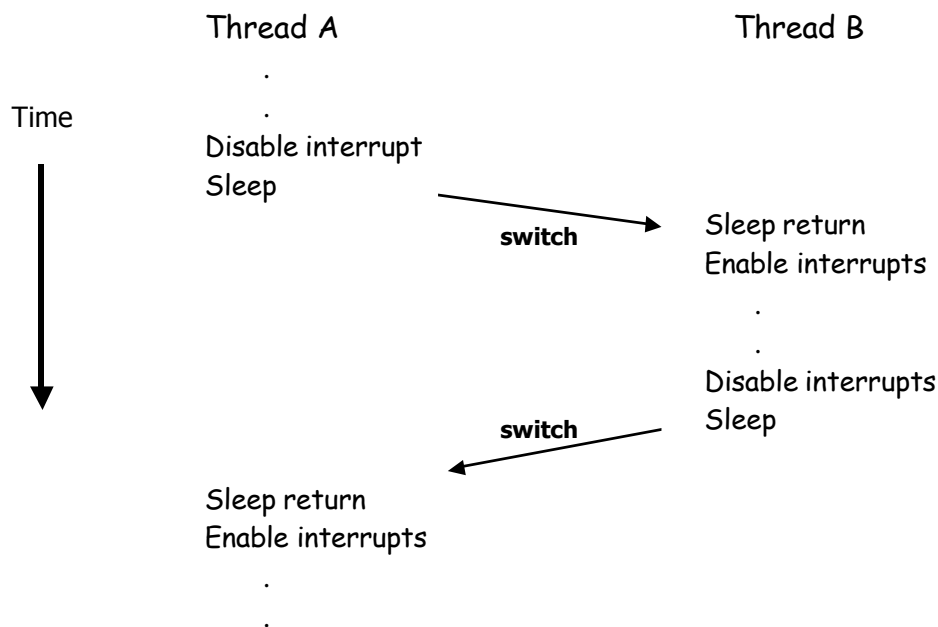
Implementing Locks: Interrupts (version 2)

```
class Lock {
public:
    void acquire();
    void release();
private:
    int value = FREE;
    Queue Q = empty;
}

Lock::acquire() {
    disable interrupts;
    if (value == FREE) {
        value = BUSY;
    } else {
        add curThread to Q;
        put curThread to sleep;
    }
    enable interrupts;
}

Lock::release() {
    disable interrupts;
    if queue not empty {
        take thread T off Q;
        put T on ready queue;
    } else {
        value = FREE;
    }
    enable interrupts;
}
```

Interrupt Disable/Enable Pattern



Implementing Locks: Atomic Spinlock

```
class Lock {
public:
    void acquire();
    void release();
private:
    int value = FREE;
}

Lock::acquire() {
    while (test&set(value) == BUSY) {
        // do nothing
    }
}

Lock::release() {
    value = FREE;
}
```

Minimizing Busy-Waiting

```
class Lock {
public:
    void acquire();
    void release();
private:
    int value = FREE;
    int guard = 0;
    Queue Q = empty;
}

Lock::acquire() {
    while (test&set(guard) == 1) {
        // do nothing
    }
    if (value == FREE) {
        value = BUSY;
        guard = 0;
    } else {
        put curThread on Q;
        guard = 0 & put curThread to sleep;
    }
}

Lock::release() {
    while (test&set(guard) == 1) {
        // do nothing
    }
    if Q is not empty {
        take T off Q;
        put T on ready queue;
    } else {
        value = FREE;
    }
    guard = 0;
}
```

Producer/Consumer

```
class ProducerConsumer {  
  
    private Queue<Item> queue;  
    private Lock lock;  
  
    public void add(Item item) {  
        lock.acquire();  
        queue.add(item);  What if full?  
        lock.release();  
    }  
  
    public Item remove() {  
        lock.acquire();  
        Item item = queue.remove();  What if empty?  
        lock.release();  
        return item;  
    }  
}
```

Condition Variables

```
private Queue<Item> queue;  
private Lock lock;  
private ConditionVariable cv;  
  
public void add(Item item) {  
    lock.acquire();  
  
    queue.add(item);  
    cv.signal(lock);  
  
    lock.release();  
}  
  
public Item remove() {  
    lock.acquire();  
  
    while (queue.isEmpty()) {  
        // release lock & sleep  
        cv.wait(lock);  
    }  
    Item item = queue.remove();  
  
    lock.release();  
    return item;  
}
```

Mesa semantics 

Condition Variables (2)

```
private Queue<Item> queue;
private Lock lock;
private ConditionVariable cv1, cv2;

public void add(Item item) {
    lock.acquire();

    while (queue.isFull()) {
        cv2.wait(lock);
    }
    queue.add(item);
    cv1.signal(lock);

    lock.release();
}

public Item remove() {
    lock.acquire();

    while (queue.isEmpty()) {
        cv1.wait(lock);
    }
    Item item = queue.remove();
    cv2.signal(lock);

    lock.release();
    return item;
}
```

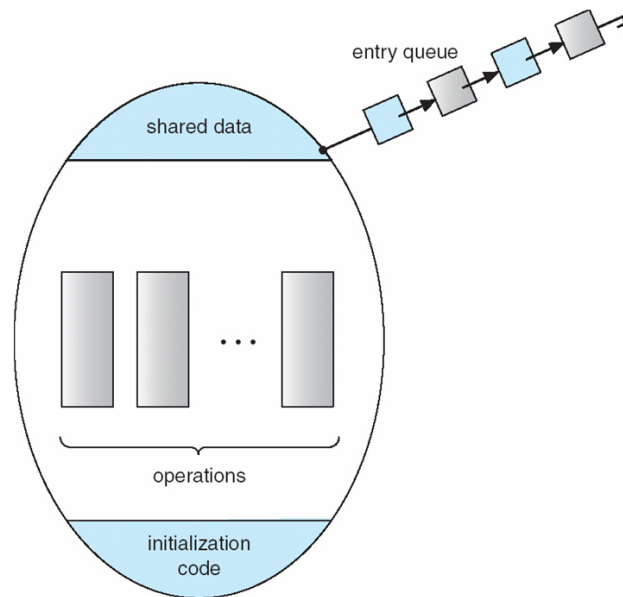
Implementing Condition Variables

```
class ConditionVariable {
public:
    void wait(Lock cvLock);
    void signal();
private:
    Queue Q = empty;
    Lock Qlock = FREE;
}

ConditionVariable::wait(Lock cvLock) {
    Qlock.acquire();
    put curThread on Q;
    Qlock.release();
    cvLock.release() & put curThread to sleep;
    cvLock.acquire();
}

ConditionVariable::signal() {
    Qlock.acquire();
    if Q is not empty {
        take T off Q;
        put T on ready queue;
    }
    Qlock.release();
}
```

Monitors



Producer/Consumer (redux)

```
class ProducerConsumer {  
  
    private Queue<Item> queue;  
    private Lock lock;  
  
    public void add(Item item) {  
        lock.acquire();  
        queue.add(item);  
        lock.release();  
    }  
  
    public Item remove() {  
        lock.acquire();  
        Item item = queue.remove();  
        lock.release();  
        return item;  
    }  
}
```

Producer/Consumer with Java Monitors

```
class ProducerConsumer {  
  
    private Queue<Item> queue;  
  
    public synchronized void add(Item item) {  
        queue.add(item);  
    }  
  
    public synchronized Item remove() {  
        return queue.remove();  
    }  
  
}
```

Java Condition Variables

```
class ProducerConsumer {  
  
    private Queue<Item> queue;  
  
    public synchronized void add(Item item) {  
        queue.add(item);  
        this.notify(); // signal  
    }  
  
    public synchronized Item remove() {  
        while (queue.isEmpty()) {  
            this.wait(); // release lock and sleep  
        }  
        return queue.remove();  
    }  
  
}
```

Too Few Chairs

Threads A, B, C, ...

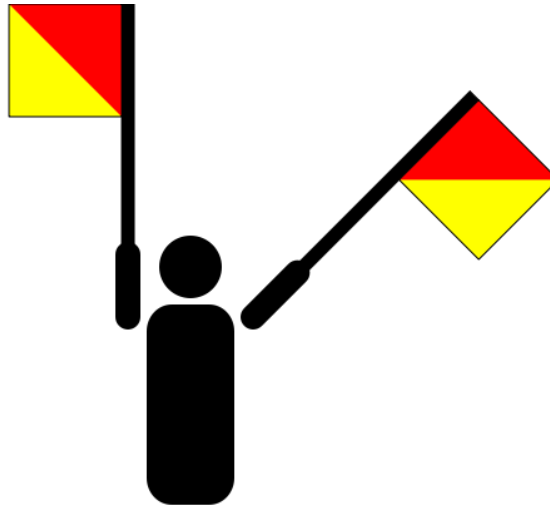
```
if (chairs > 0) {  
    chairs--;  
    doSomeWork();  
    chairs++;  
}
```

Too Few Chairs with Condition Variables

Threads A, B, C, ...

```
lock.acquire();  
if (chairs == 0) {  
    cv.wait(lock);  
}  
chairs--;  
lock.release();  
doSomeWork();  
lock.acquire();  
chairs++;  
cv.signal(lock);  
lock.release();
```

Semaphores



Counting Semaphore

```
sem = new Semaphore(NUM_CHAIRS);
```

Threads A, B, C, ...

```
sem.wait();  
doSomeWork();  
sem.signal();
```

Binary Semaphore

```
sem = new Semaphore(1);
```

Threads A, B, C, ...

```
sem.wait();  
if (noMilk) {  
    buy milk;  
}  
sem.signal();
```

Semaphore for Ordering

```
sem = new Semaphore(0);
```

Thread A

```
attend_os_class();  
write_notes();  
sem.signal();
```

Thread B

```
sem.wait();  
read_notes();  
do_os_project();
```

Condition Variables (redux)

```
private Queue<Item> queue;
private Lock lock;
private ConditionVariable cv1, cv2;

public void add(Item item) {
    lock.acquire();

    while (queue.isFull()) {
        cv2.wait(lock);
    }
    queue.add(item);
    cv1.signal(lock);

    lock.release();
}

public Item remove() {
    lock.acquire();

    while (queue.isEmpty()) {
        cv1.wait(lock);
    }
    Item item = queue.remove();
    cv2.signal(lock);

    lock.release();
    return item;
}
```

Producer/Consumer with Semaphores

```
private Queue<Item> queue(N); // capacity N
private Semaphore mutex = 1; // binary (lock)
private Semaphore empty = N; // # free slots
private Semaphore full = 0; // # occupied slots

public void add(Item item) {
    // one fewer slot, or wait
    empty.wait();

    mutex.wait();
    queue.add(item);
    mutex.signal();

    // one more used slot
    full.signal();
}

public Item remove() {
    // wait until nonempty
    full.wait();

    mutex.wait();
    Item item = queue.remove();
    mutex.signal();

    // one more free slot
    empty.signal();

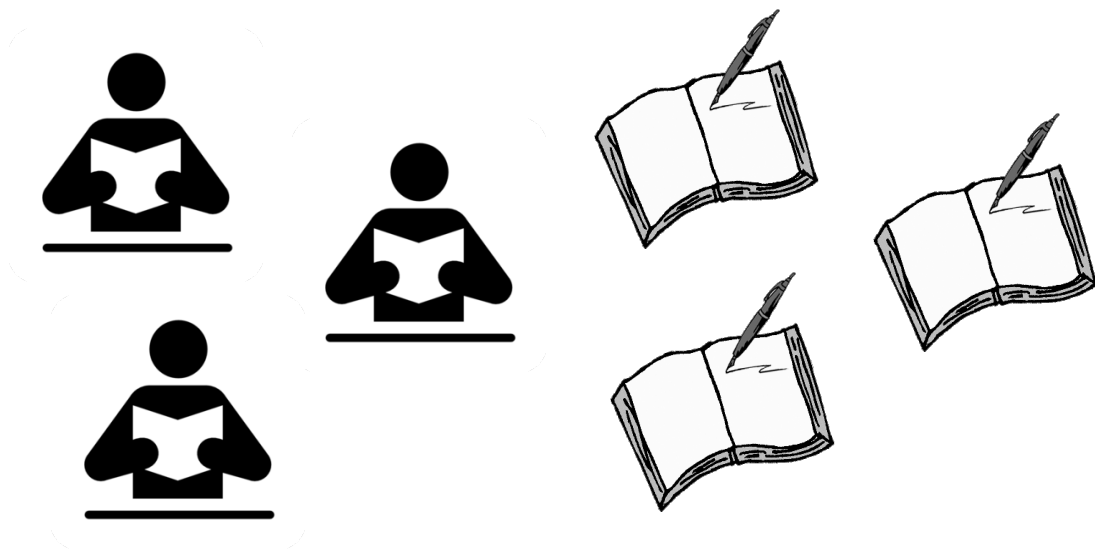
    return item;
}
```

Implementing Semaphores

```
class Semaphore {  
    public:  
        Semaphore(int N); // starting value  
        void wait();  
        void signal();  
    private:  
        int value = N;  
        Queue Q = empty;  
}  
  
Semaphore::wait() {  
    value--;  
    if (value < 0) {  
        put curThread on Q;  
        put curThread to sleep;  
    }  
}  
  
Semaphore::signal() {  
    value++;  
    if (value <= 0) {  
        take T off Q;  
        put T on ready queue;  
    }  
}
```

(+ atomicity via interrupts or Test&Set)

Readers/Writers Problem



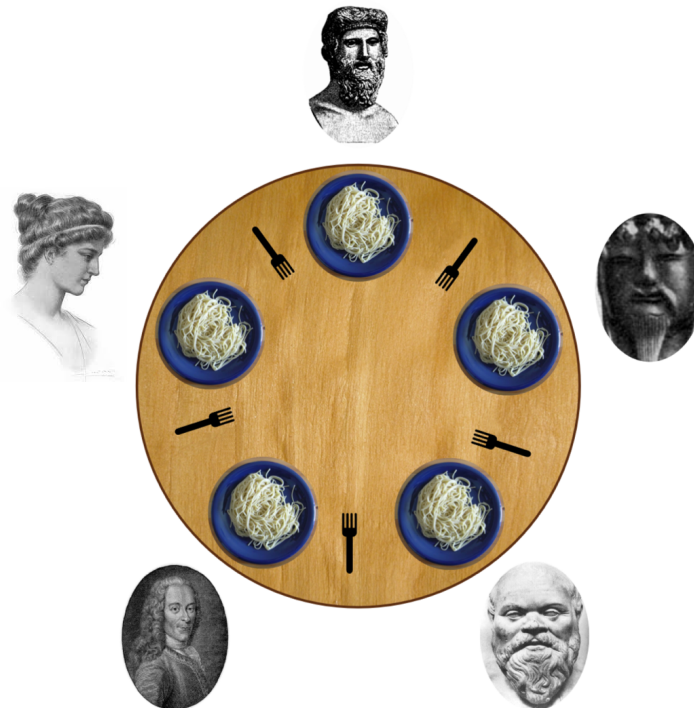
Readers/Writers with Semaphores

```
class ReadWrite {  
    public:  
        void read();  
        void write();  
    private:  
        Semaphore wrt = 1;  
        Semaphore mutex = 1;  
        int readers = 0;  
}  
  
ReadWrite::write() {  
    wrt.wait();  
    <perform write>  
    wrt.signal();  
}  
  
ReadWrite::read() {  
    mutex.wait();  
    readers++;  
    if (readers == 1)  
        wrt.wait();  
    mutex.signal();  
    <perform read>  
    mutex.wait();  
    readers--;  
    if (readers == 0)  
        wrt.signal();  
    mutex.signal();  
}
```

Readers/Writers with Monitors

```
private int numReaders = 0;  
private int numWriters = 0;  
  
private synchronized void prepareRead() {  
    while (numWriters > 0) wait();  
    numReaders++;  
}  
  
private synchronized void doneRead() {  
    numReaders--;  
    if (numReaders == 0) notify();  
}  
  
public void read() {  
    // reads NOT synchronized  
    prepareRead();  
    <perform read>  
    doneRead();  
}  
  
private void prepareWrite() {  
    numWriters++;  
    while (numReaders > 0) wait();  
}  
  
private void doneWrite() {  
    numWriters--;  
    notify();  
}  
  
public synchronized void write() {  
    // writes synchronized  
    prepareWrite();  
    <perform write>  
    doneWrite();  
}
```

The Dining Philosophers



Dining Philosophers with Locks

```
Lock chopsticks[5];

void philosopher(int i) {
    while (true) {

        think();

        chopsticks[i].acquire(); // left chopstick
        chopsticks[(i+1)%5].acquire(); // right chopstick

        eat();

        chopsticks[i].release();
        chopsticks[(i+1)%5].release();
    }
}
```

**Possible
Deadlock!**

Dining Philosophers with Monitors

```
monitor DiningPhilosophers {
    enum {EAT, THINK, HUNGRY}
    state[5];
    condition self[5];
}

void synchronized pickup(int i) {
    state[i] = HUNGRY;
    tryEat(i);
    if (state[i] != EAT)
        self[i].wait();
}

void synchronized putdown(int i) {
    state[i] = THINK;
    // test left and right neighbors
    tryEat((i+4)%5);
    tryEat((i+1)%5);
}

void philosopher(int i) {
    state[i] = THINK;
    while (true) {
        think();
        pickup(i); // pickup both
        eat();
        putdown(i);
    }
}

void tryEat(int n) {
    // check left and right of n
    if (state[n] == HUNGRY &&
        state[(n+4)%5] != EAT &&
        state[(n+1)%5] != EAT) {
        state[n] = EAT;
        self[n].signal();
    }
}
```

Deadlock

Thread A:

```
lock1.acquire();
lock2.acquire();

// do something

lock1.release();
lock2.release();
```

Thread B:

```
lock2.acquire();
lock1.acquire();

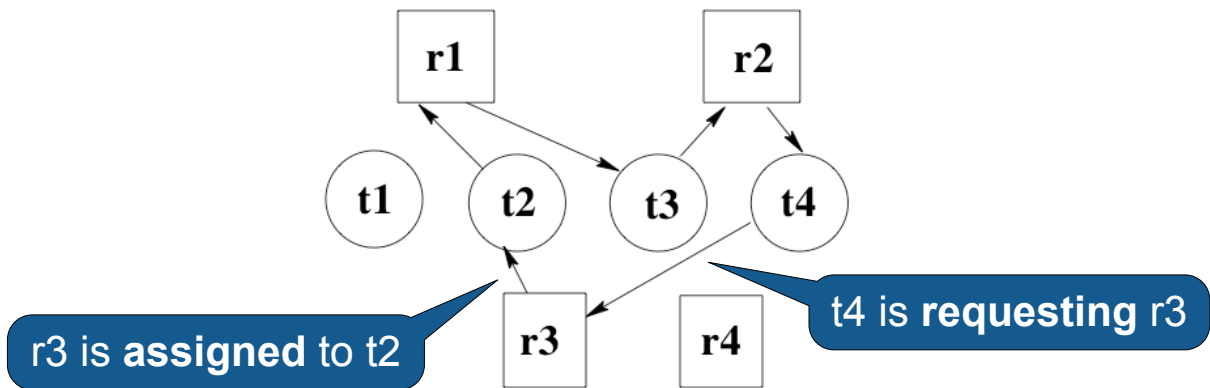
// do something

lock1.release();
lock2.release();
```

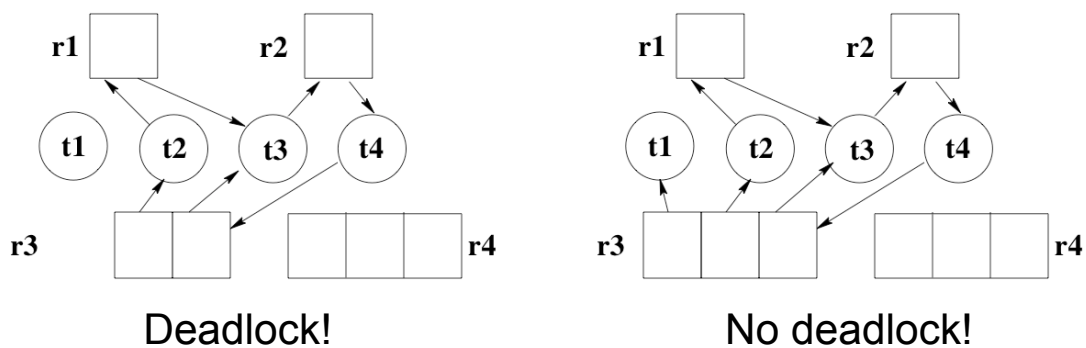
Thread group waiting on events from same group

1. Resources have **mutual exclusion**
2. Threads **hold and wait**
3. **No preemption** of resources
4. **Circular wait**

Resource Allocation Graph



Multiple Copies of Resources



Allocation Example

3 threads, 12 interchangeable resources (11 used)

	maximum	current	could request
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

Safe state!

Allocation Example

t_3 requests last resource (now all 12 used)

	maximum	current	could request
t_1	4	3	1
t_2	8	4	4
t_3	12	5	7

Unsafe state!

Deadlock Avoidance with RAG

